

---

# Resumen Organización del Computador II

Axel Straminsky

Diciembre 2014



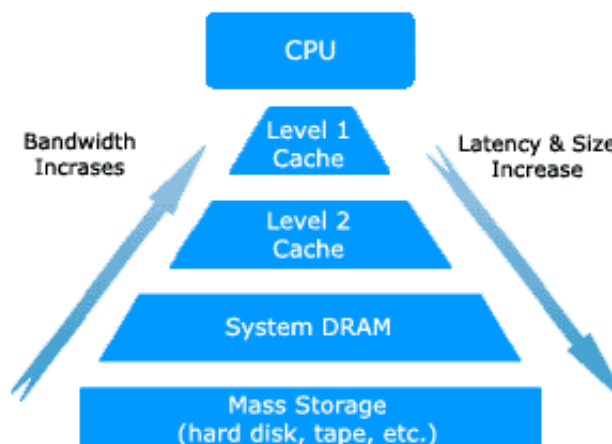
**DEPARTAMENTO  
DE COMPUTACION**

Facultad de Ciencias Exactas y Naturales - UBA

# Índice

<b>1. Jerarquía de Memoria</b>	<b>3</b>
<b>2. Memoria Cache</b>	<b>3</b>
2.1. Mapeo Directo . . . . .	4
2.2. Mapeo Totalmente Asociativo . . . . .	4
2.3. Mapeo Asociativo por Conjuntos . . . . .	4
2.4. Algoritmos de Reemplazo . . . . .	6
2.5. Políticas de Escritura . . . . .	6
<b>3. Direccionamiento de Memoria</b>	<b>7</b>
3.1. Segmentación en Hardware . . . . .	7
3.1.1. Selectores de segmento y registros de segmentación . . . . .	7
3.1.2. Descriptores de Segmento . . . . .	7
3.1.3. Paginación en Hardware . . . . .	8
3.1.4. Paginación Regular . . . . .	8
3.1.5. Paginación Extendida . . . . .	9
3.1.6. Physical Address Extension (PAE) . . . . .	9
<b>4. Modos de Operación</b>	<b>10</b>
<b>5. Taxonomía de Flynn</b>	<b>11</b>
<b>6. Symmetric Multiprocessors (SMP)</b>	<b>12</b>
6.1. Coherencia de Cache . . . . .	12
6.1.1. Soluciones por Software . . . . .	12
6.1.2. Soluciones por Hardware . . . . .	12
6.2. Protocolo MESI . . . . .	13
<b>7. Instruction Pipelining</b>	<b>14</b>
7.1. Branches . . . . .	14
7.2. Hazards . . . . .	15
<b>8. Procesadores Superescalares</b>	<b>16</b>
8.1. Out of Order Execution . . . . .	16
8.2. Scoreboarding . . . . .	16
8.3. Tomasulo . . . . .	17
<b>9. Bibliografía</b>	<b>18</b>

## 1. Jerarquía de Memoria



El objetivo de esta jerarquía es aprovechar la referencia de *localidad espacial* y la *referencia de temporalidad*.

La primera asume que si se accedió a un dato, probablemente se vayan a acceder a datos contiguos; la segunda asume que un dato accedido se va a volver a acceder pronto. Generalmente, la mayoría de los accesos a un subsistema de memoria se hacen de manera transparente entre un nivel de la jerarquía de memoria y el nivel inmediatamente superior o inferior. Por ejemplo, la CPU rara vez accede directamente a la memoria principal, sino que cuando la CPU requiere datos de memoria, el subsistema de cache L1 se hace cargo. Si los datos requeridos se encuentran en la cache L1, entonces ésta entrega los datos a la CPU. En otro caso, el subsistema de cache L1 pasa el pedido al subsistema de cache L2. Si los datos se encuentran en la cache L2, estos se pasan a la cache L1, y de la cache L1 a la CPU. Luego, si la CPU vuelve a solicitar estos datos, los encontrará en la cache L1.

Generalmente, los subsistemas de memoria mueven bloques de datos, o líneas de cache, cada vez que acceden a niveles inferiores de la jerarquía. Por ejemplo, si se ejecuta `mov eax, mem32`, y el valor de `mem32` no está en la cache L1, el controlador de la cache no lee simplemente los 32 bits de `mem32` de la cache L2, asumiendo que se encuentren ahí. En cambio, el controlador de la cache va a leer un bloque de bytes (16, 32 o 64) de la cache L2. Se espera que el programa exhiba localidad espacial y que por lo tanto los futuros accesos sean más rápidos.

## 2. Memoria Cache

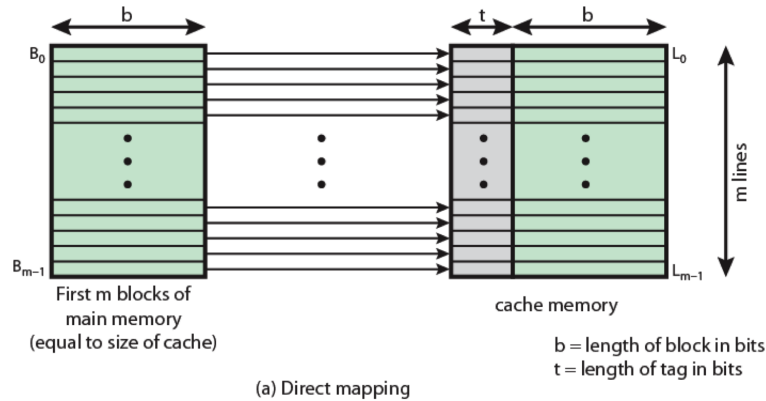
La memoria cache no se encuentra organizada como un conjunto de bytes, sino que se encuentra organizada en líneas de cache, donde cada línea contiene 16, 32 o 64 bytes.

Existen tres esquemas de cache diferentes: cache de mapeo directo, cache totalmente asociativa y cache asociativa en conjuntos de  $n$  vías.

La memoria principal consiste de  $2^n$  posiciones direccionables, con cada posición teniendo una dirección única de  $n$  bits. Para los propósitos del mapeo, se considera que esta memoria consiste de un conjunto de bloques de longitud fija de  $k$  direcciones cada uno. Por lo tanto, hay  $M = \frac{2^n}{k}$  bloques en memoria principal, y la cache consiste en  $m$  líneas, cada una de las cuales contiene  $k$  direcciones, más unos bits de tag y unos bits de control.

La cantidad de líneas es considerablemente menor que la cantidad de bloques en memoria principal ( $m \ll M$ ). Si una palabra en un bloque de memoria principal es leída, entonces todo el bloque es transferido a una de las líneas de cache. Como varios bloques distintos pueden ser copiados a una misma línea, se incluye un tag que identifica qué bloque en particular se encuentra actualmente en la línea.

### 2.1. Mapeo Directo



En este esquema, cada bloque de memoria principal es mapeado a una única posible línea de cache. El mapeo se expresa como  $i = j \bmod m$ , donde

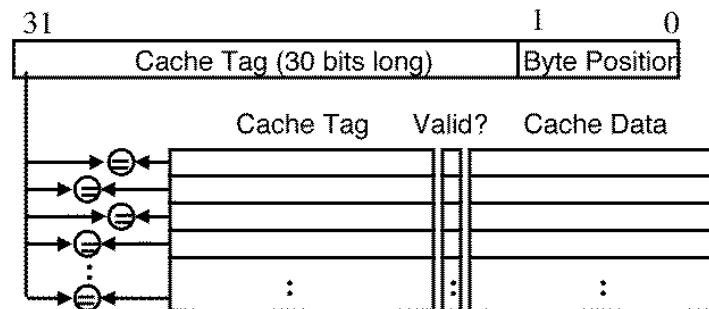
$i$  = número de la línea de cache

$j$  = número de bloque de memoria principal

$m$  = cantidad de líneas de la cache

Una desventaja de esto es que 2 bloques que mapean a la misma posición se pueden estar continuamente pisando, sin usar el resto de la cache.

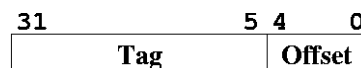
### 2.2. Mapeo Totalmente Asociativo



El mapeo totalmente asociativo elimina la desventaja del mapeo directo permitiendo que un bloque de memoria principal sea cargado en cualquier línea de la cache. Para determinar si un bloque se encuentra en la cache, la lógica de control de la cache debe examinar simultáneamente el tag de todas las líneas para encontrar un match. En este esquema hay cierta flexibilidad en cuanto a que bloque descartar cuando la cache está llena y se lee un nuevo bloque.

La principal desventaja de este enfoque es la complejidad de los circuitos que se muestran para examinar todos los tags de la cache en paralelo.

Las direcciones de memoria se dividen de la siguiente manera:



### 2.3. Mapeo Asociativo por Conjuntos

En este caso, la cache consiste en un número de conjuntos, cada uno de los cuales consiste en un número de líneas. Las relaciones son:

$$m = v * k$$

$$i = j \bmod k$$

donde

$i$  = número del conjunto

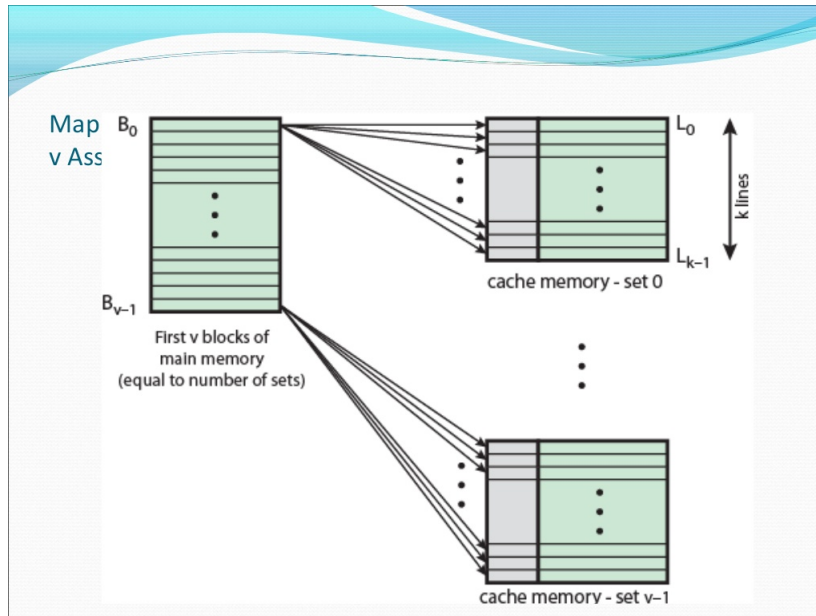
$j$  = número del bloque de memoria principal

$m$  = cantidad de líneas en la cache

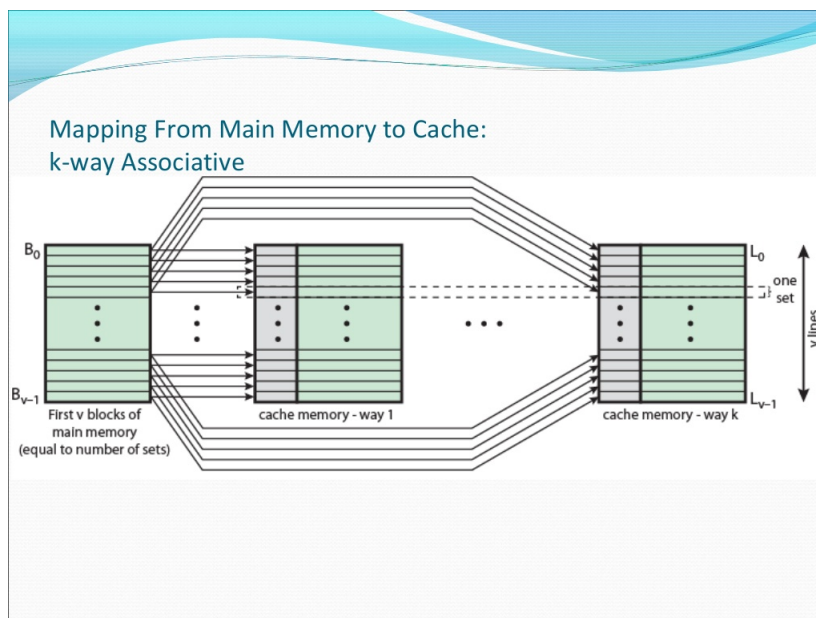
$v$  = cantidad de conjuntos

$k$  = cantidad de líneas en cada conjunto

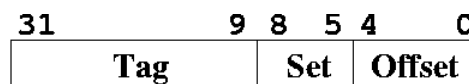
Esto se denomina cache asociativa por conjuntos de  $k$ -vías. En el mapeo asociativo por conjuntos, cada palabra se puede mapear a cualquier línea de cache en un conjunto específico. Por lo tanto, la cache asociativa por conjuntos se puede implementar físicamente como  $v$  caches asociativas.



También se pueden implementar como  $k$  caches de mapeo directo. Cada cache de mapeo directo se la denomina una vía, que consiste en  $v$  líneas. Las primeras  $v$  líneas de memoria principal se mapean directamente en las  $v$  líneas de la vía, y así sucesivamente.



En este caso, la controladora de la cache interpreta las direcciones de memoria como 3 campos: Tag, Set y Offset.



## 2.4. Algoritmos de Reemplazo

Una vez que la cache está llena, si se trae un nuevo bloque a la cache, se debe descartar uno de los bloques existentes. Para hacer esto, se debe implementar un algoritmo en hardware. Uno de los más efectivos es LRU (least recently used): reemplazar en la cache el bloque que hace más tiempo que no es usado. Otra posibilidad es usar un algoritmo FIFO, y reemplazar el bloque que hace más tiempo que está en la cache. Por último, se puede utilizar un algoritmo de LFU (least frequently used), que reemplaza el bloque que experimentó la menor cantidad de referencias.

## 2.5. Políticas de Escritura

Cuando un bloque de la cache es reemplazado, existen 2 casos a considerar: si el bloque viejo no fue alterado, entonces puede ser reemplazado sin más; en otro caso, la memoria principal debe ser actualizada antes de descartar el bloque. Existen dos problemas con respecto a esto. Primero, más de un dispositivo puede tener acceso a la memoria principal, por ejemplo un dispositivo de E/S. Si una palabra ha sido alterada en la cache, entonces la palabra correspondiente en memoria es inválida, y si el dispositivo de E/S alteró una palabra en la memoria principal, entonces esa palabra en la cache es inválida.

Otro problema que ocurre es cuando se tienen varios procesadores compartiendo el mismo bus y cada uno tiene su propia cache. Entonces, si una palabra es alterada en una cache, puede invalidar el resto de las caches.

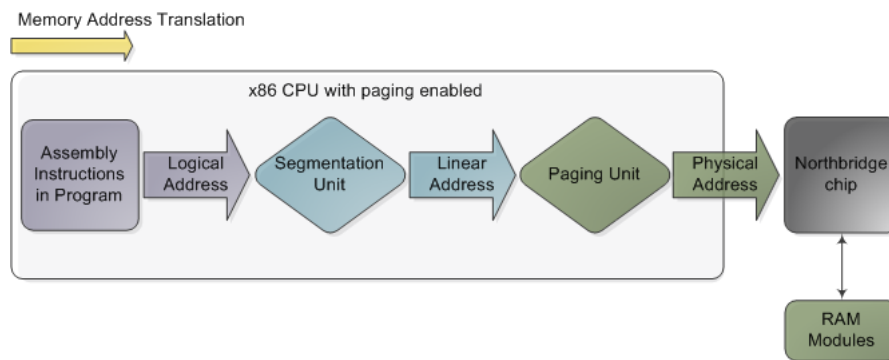
La técnica más simple es el **write through**. Con esta técnica, todas las operaciones de escritura son hechas a la memoria principal y a la cache, asegurando que la memoria principal siempre es válida. La desventaja principal de esta técnica es que genera mucho tráfico de memoria, lo cual puede ocasionar un bottleneck. Con **write through buffered**, el procesador actualiza la cache, y el controlador cache luego actualiza la copia en memoria DRAM mientras el procesador continúa ejecutando instrucciones y usando datos de la memoria cache.

Otra técnica, conocida como **write back**, minimiza las escrituras a memoria, ya que las actualizaciones se hacen solo en la cache. Cuando se hace una actualización, se prende un bit asociado con la línea de cache llamado dirty bit. Entonces, cuando un bloque es reemplazado, es escrito en memoria principal si y solo si el bit dirty está encendido. El problema con esta política es que porciones de la memoria principal pueden ser inválidas, y por lo tanto accesos de dispositivos de E/S solo se pueden realizar a través de la cache, lo cual puede crear un bottleneck.

### 3. Direccionamiento de Memoria

En los procesadores 80x86 existen 3 tipos distintos de direcciones.

- Direcciones lógicas: están relacionadas con la arquitectura de segmentos de los 80x86. Cada dirección lógica consiste en un segmento y en un offset, que denota la distancia desde el inicio del segmento hasta la dirección buscada.
- Direcciones lineales (o virtuales): un entero sin signo de 32 bits que puede direccionar hasta 4GB.
- Direcciones físicas: son las direcciones físicas de la memoria.

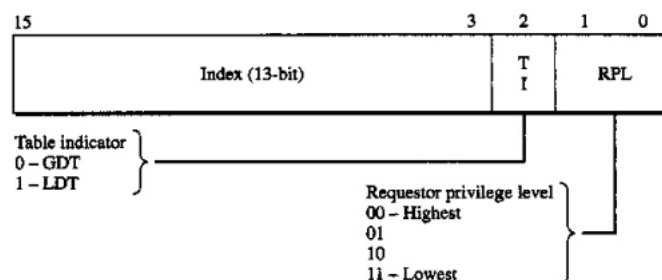


#### 3.1. Segmentación en Hardware

##### 3.1.1. Selectores de segmento y registros de segmentación

Una dirección lógica consiste de dos partes: un identificador de segmento y un offset que especifica la dirección relativa dentro del segmento. El identificador de segmento es un campo de 16 bits llamado **selector de segmento**, mientras que el offset es un campo de 32 bits.

El procesador provee **registros de segmentación** cuyo único propósito es guardar los selectores de segmento. Los registros son: *cs*, *ss*, *ds* de propósito específico, y *es*, *fs*, *gs* de propósito general. El registro *cs* tiene otra función importante: incluye un campo de 2 bits que especifica el *Request Privilege Level* (RPL) de la CPU.



El índice identifica la entrada del descriptor de segmento en la GDT o LDT. Como los descriptors de segmento son de 8 bytes, el índice se debe multiplicar por 8.

El TI (Table Indicator) especifica si el descriptor de segmento está en la GDT (TI = 0) o en la LDT (TI = 1).

El RPL especifica el CPL de la CPU cuando el selector de segmento es cargado en el registro *cs*.

Como Linux prácticamente no utiliza segmentación salvo donde lo requiera el procesador, el offset de una dirección lógica coincide siempre con la dirección lineal.

##### 3.1.2. Descriptores de Segmento

Cada segmento está representado por un **descriptor de segmento** de 8 bytes, el cual está guardado en la *Global Descriptor Table* (GDT) o en la *Local Descriptor Table* (LDT).

Existen varios tipos de descriptors de segmento:

- Descriptor de segmento de código: indica que el descriptor de segmento se refiere a un segmento de código. Puede estar en la GDT o en la LDT.

- Descriptor de segmento de datos: indica que el descriptor de segmento se refiere a un segmento de datos. Puede estar en la GDT o en la LDT. Un segmento de stack se implementa mediante un segmento de datos genérico.
- Descriptor de segmento de estado de tarea: indica que el descriptor de segmento se refiere a un Task State Segment (TSS), es decir, un segmento usado para guardar los contenidos de los registros del procesador. Puede estar solo en la GDT.

Cada vez que un selector de segmento se carga en un registro de segmento, el correspondiente descriptor de segmento es cargado en un registro especial que no es accesible al programador. De esta manera, las traducciones de direcciones lógicas referidas a ese segmento se pueden realizar sin pasar por la GDT. Este registro se actualiza cada vez que cambia el selector de segmento.

### 3.1.3. Paginación en Hardware

La unidad de paginación traduce las direcciones lineales en direcciones físicas. Una función fundamental de la unidad es chequear el tipo de acceso requerido contra los derechos de acceso de la dirección lineal; si el acceso a memoria no es válido, genera un Page Fault.

Por una cuestión de eficiencia, las direcciones lineales están agrupadas en intervalos de longitud fija llamados páginas; direcciones lineales continuas dentro de una página están mapeadas en direcciones físicas continuas.

De esta manera, el kernel puede especificar la dirección física y los derechos de acceso de una página en vez de los de las direcciones lineales incluidas en ella.

Las estructuras de datos que mapean direcciones lineales a físicas son llamadas page tables. Están guardadas en memoria principal y deben ser inicializadas antes de habilitar la paginación.

### 3.1.4. Paginación Regular

La traducción de la dirección lineal se realiza en 2 pasos, cada uno basado en una tabla de traducción. El objetivo de este esquema en dos niveles es reducir la cantidad de memoria RAM necesaria para las page tables de cada proceso. Si se utilizase un solo nivel de tablas, se necesitarían  $2^{20}$  entradas (4 bytes por entrada = 4 MB) para representar las page tables de cada proceso. El esquema con 2 niveles reduce la cantidad de memoria necesaria ya que requiere page tables solo para aquellas regiones de memoria virtual realmente usadas por el proceso.

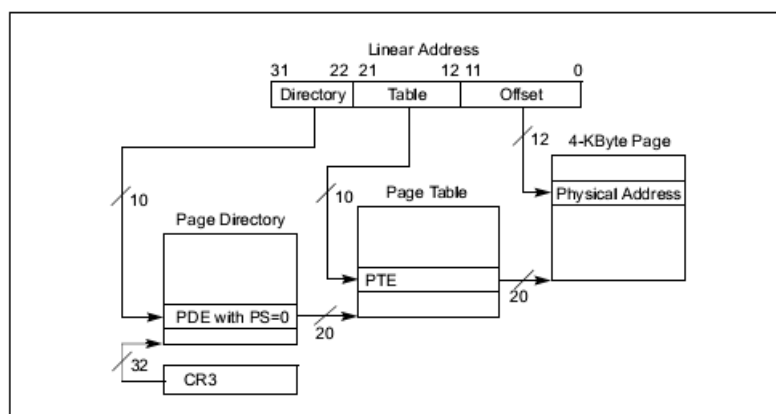
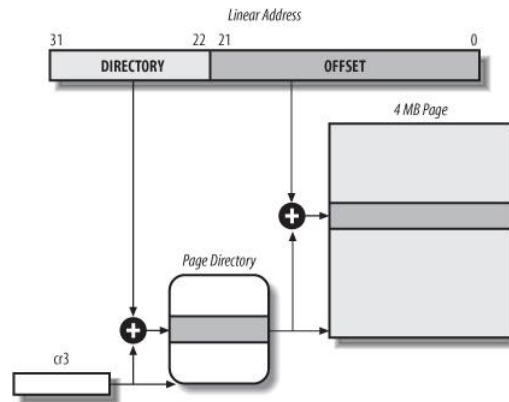


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging



### 3.1.5. Paginación Extendida

En la paginación extendida, las páginas son de 4 MB en vez de 4 KB.



### 3.1.6. Physical Address Extension (PAE)

La cantidad de memoria RAM que soporta un procesador está limitada por el número de pins de direcciones conectados al bus de direcciones. Los procesadores del 80386 hasta el Pentium direccionan hasta 4 GB de memoria.

Como algunas computadoras necesitaban más memoria (dentro de la arquitectura 80x86 de 32 bits), Intel aumentó el número de los pins de direcciones de 32 a 36. Esto puede ser explotado con un nuevo sistema de paginación que permita traducir direcciones lineales de 32 bits en direcciones físicas de 36 bits.

## 4. Modos de Operación

Los procesadores IA-32 tienen 3 modos de operación:

- Modo Real: en este modo el procesador implementa el entorno de operación del 8086 con algunas extensiones: puede pasar por software a Modo Protegido o a Modo Mantenimiento del Sistema y utilizar registros de 32 bits.
- Modo Protegido: en este modo se implementa multitasking y se despliega un espacio de direccionamiento de 4 GB, extensible a 64 GB. A partir del 80386 se introduce un sub modo al que puede ponerse a una determinada tarea, denominado Virtual-8086, que permite a un programa diseñado para ejecutarse en 8086, poder ejecutarse como una tarea en Modo Protegido.
- Modo Mantenimiento del Sistema: este modo fue introducido para realizar funciones específicas para la plataforma de hardware en la cual se desempeña el procesador, como ahorro de energía y seguridad.

Los procesadores Intel 64 además de los modos de trabajo de los IA-32 incluyen un modo IA-32e, al que se pasa estando en Modo Protegido, con paginación habilitada y PAE activo. En este modo existen a su vez 2 sub modos: Compatibilidad y Modo 64 bits.

- Modo Compatibilidad: este modo permite a las aplicaciones de 16 y 32 bits ejecutarse sin recompilación bajo un sistema operativo de 64 bits. El entorno de ejecución es el de la arquitectura IA-32, y no soporta el manejo de tareas del modo IA-32 (TSS mediante) ni el modo Virtual-8086.
- Modo 64 bits: este modo habilita al sistema operativo de 64 bits a ejecutar tareas utilizando direcciones lineales de 64 bits.

## 5. Taxonomía de Flynn

Es la manera más común de categorizar a los sistemas con capacidad de procesamiento paralelo.

- Single Instruction, Single Data (SISD): un solo procesador ejecuta un solo flujo de instrucciones que operan en datos almacenados en una sola memoria.
- Single Instruction, Multiple Data (SIMD): una sola instrucción de máquina controla la ejecución simultánea de un número de elementos.
- Multiple Instruction, Single Data (MISD): no existen muchas arquitecturas que lo implementen.
- Multiple Instruction, Multiple Data (MIMD): CLusters, SMP, NUMA.

## 6. Symmetric Multiprocessors (SMP)

Un SMP puede definirse como un sistema con las siguientes características:

- Hay 2 o más procesadores de capacidad similar.
- Estos procesadores comparten la misma memoria principal y dispositivos de E/S, y están interconectados por un bus, de manera tal que el tiempo de acceso a la memoria es aproximadamente el mismo para cada procesador.
- Todos los procesadores pueden realizar las mismas operaciones (de ahí simétrico).
- Un sistema operativo que provea interacción entre los procesadores y las tareas

Cada procesador tiene acceso a la memoria principal y a los dispositivos de E/S a través de algún mecanismo de intercomunicación. Los procesadores se pueden comunicar entre sí mediante memoria o intercambiando señales directamente.

La organización más común es la del bus de tiempo compartido. La estructura y las interfaces son básicamente las mismas que para un sistema de un solo procesador. El bus consiste en líneas de control, direcciones y datos. La desventaja principal es la performance, ya que como todas las referencias a memoria pasan a través de un bus común, la velocidad del bus limita la velocidad de todo el sistema.

Para mejorar la performance, cada procesador cuenta con una memoria cache interna. Esto introduce nuevos problemas, ya que si un dato es modificado en una cache, puede invalidar al mismo dato en otra cache.

Este problema es conocido como el problema de la coherencia de cache, y la solución típicamente involucra cambios de hardware.

### 6.1. Coherencia de Cache

En los sistemas multiprocesador, es común que cada procesador tenga varios niveles de cache, aunque esto crea un problema llamado “coherencia de cache”. La esencia del problema es esta: múltiples copias del mismo dato pueden existir en diferentes caches simultáneamente, y si cada procesador actualiza su copia, se tiene una visión inconsistente de la memoria. Dos políticas comunes de escritura en caches son:

- Write Back: las operaciones de escritura se hacen solo a la cache. La memoria principal se actualiza cuando la línea de cache se remueve de la cache.
- Write Through: todas las operaciones de escritura se realizan tanto en la cache como en la memoria principal.

Es claro que la política de Write Back puede resultar en una inconsistencia. Si 2 caches contienen la misma línea, y la línea se actualiza en una cache, la otra cache contendrá un valor inválido. Incluso con la política de Write Through pueden ocurrir inconsistencias a menos que las otras caches reciban una notificación de la actualización.

#### 6.1.1. Soluciones por Software

Los esquemas de coherencia de cache por software dependen del compilador y el sistema operativo para tratar con el problema.

La ventaja de este enfoque es que transfiere el overhead de detectar potenciales problemas de tiempo de ejecución a tiempo de compilación, y la complejidad de diseño del hardware al software. La desventaja es que el software muchas veces debe tomar decisiones conservadoras, lo cual lleva a una utilización ineficiente de la cache. Una posible política es marcar algunos datos como no cacheables.

#### 6.1.2. Soluciones por Hardware

Las soluciones basadas en hardware son generalmente referidas como protocolos de coherencia de cache. Estas soluciones proveen reconocimiento en tiempo de ejecución de potenciales inconsistencias. Como el problema solamente es tratado cuando realmente aparece, hay un uso más efectivo de la cache, llevando a una mejora en performance con respecto a enfoques basados en software. Además, estos enfoques son transparentes al programador.

En los esquemas de hardware existen 2 categorías:

- Protocolos de Directorio: los protocolos de directorio recolectan y mantienen información sobre donde residen las copias de las líneas. Generalmente, hay un controlador centralizado que es parte del controlador de la memoria principal, y un directorio que está almacenado en memoria principal.

El directorio contiene información de estado global sobre los contenidos de las caches. Cuando un controlador individual de cache hace un pedido, el controlador centralizado chequea y envía los comandos necesarios para transferir datos entre memoria y cache o entre caches. También es responsable de mantener la información de estado actualizada.

Generalmente, el controlador mantiene información acerca de qué procesadores tienen una copia de qué líneas. Antes de que un procesador pueda escribir a una copia local de la línea, debe pedir acceso exclusivo a esa línea al controlador. Antes de otorgar acceso exclusivo, el controlador manda un mensaje a todos los procesadores con una copia en la cache de esta línea para que la invaliden.

La desventaja de este esquema es que existe un cuello de botella entre los controladores de cache y el controlador central.

- **Protocolos Snoop:** estos protocolos distribuyen la responsabilidad de mantener la coherencia de cache entre todos los controladores de cache del sistema. Una cache debe reconocer cuando una línea que posee es compartida con otras caches. Cuando se actualiza una línea, se anuncia esto a todas las demas caches mediante un mecanismo de broadcasting. Cada controladora de cache puede “hurgar” (snoop) en la red para observar estas notificaciones y actuar en consecuencia.

Se han explorado dos enfoques: **Write Invalidate** y **Write Update**. Con un protocolo Write invalidate, pueden haber varios lectores pero un solo escritor en un momento dado. Cuando una cache quiere realizar una escritura, primero envía un mensaje que invalida esa línea en las demas caches, haciendo que la línea sea exclusiva para la cache de escritura.

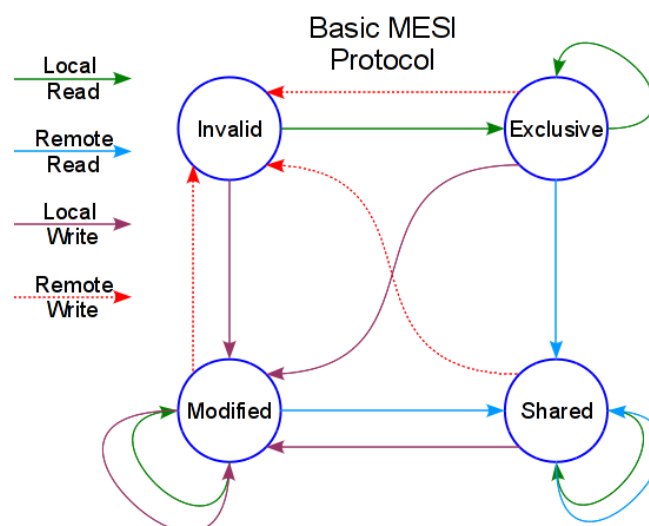
Con el protocolo Write Update, pueden haber múltiples escritores y lectores. Cuando un procesador quiere actualizar una línea compartida, la palabra a ser actualizada es distribuída a los demás procesadores.

El protocolo Write Invalidate marca el estado de cada línea de cache (usando 2 bits del tag) como modificado, exclusivo, compartido (shared) e inválido. Por esta razón, a este protocolo se lo llama **protocolo MESI**.

## 6.2. Protocolo MESI

Para este protocolo, la cache de datos incluye 2 bits de status por cada tag, por lo que cada línea puede estar en uno de cuatro estados:

- **Modificado:** la línea en la cache ha sido modificada y está disponible solo en esa cache, y es diferente al dato que está en memoria principal.
- **Exclusivo:** la línea en la cache es la misma que la que está en memoria principal y no está presente en otras caches.
- **Shared:** la línea en la cache es la misma que la que está en memoria principal y puede estar presente en otra cache.
- **Inválido:** la línea en la cache contiene datos inválidos.



## 7. Instruction Pipelining

Es una técnica utilizada para aumentar el throughput de instrucciones (cantidad de instrucciones ejecutadas por unidad de tiempo). El ciclo de instrucciones se parte en series llamadas pipelines, y cada uno de estos pasos puede ser ejecutado concurrentemente.

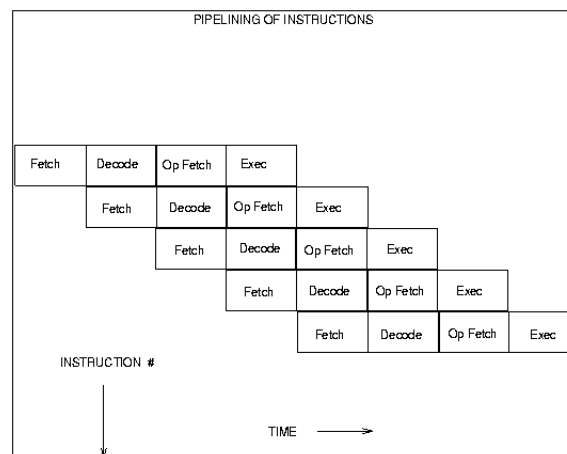
El pipelining incrementa el throughput de instrucciones al hacer múltiples operaciones concurrentemente, pero no reduce la latencia de las instrucciones (el tiempo que tardan en completarse). De hecho, puede aumentar la latencia debido al overhead de dividir las instrucciones, o debido a que el pipeline debe pararse.

El objetivo es que el procesador trabaje con tantas instrucciones a la vez como pasos independientes hay en una instrucción.

Hay ciertos factores que limitan la performance; por ejemplo, si la duración de las etapas no es igual, otras partes del pipeline van a tener que quedarse en espera. Otra dificultad es el branching condicional de las instrucciones, que puede invalidar muchas instrucciones levantadas. Otro evento impredecible es una interrupción.

En principio parece que mientras mayor número de etapas haya en un pipeline, mayor es el ritmo de ejecución. Sin embargo, esto no es así por 2 motivos:

- En cada etapa del pipeline hay cierto overhead en mover los datos entre los distintos buffers. Esto es significativo cuando las instrucciones secuenciales son lógicamente dependientes, ya sea por el uso de branching o por el acceso a memoria.
- La cantidad de lógica de control requerida para manejar dependencias de memoria y registros y para optimizar el uso del pipeline aumenta enormemente con el número de etapas.



### 7.1. Branches

Uno de los mayores problemas al diseñar un pipeline es asegurarse que continuamente lleguen instrucciones a las etapas iniciales. El principal impedimento para esto es el branching condicional, y existen varios enfoques para tratarlo:

- Flujos Múltiples: consiste en replicar las porciones iniciales del pipeline y levantar instrucciones de ambos branches
- Loop Buffer: consiste en una pequeña memoria muy rápida, que es mantenida por la etapa del fetch de instrucciones del pipeline, y contiene las  $n$  instrucciones secuenciales más recientemente levantadas. Si se tiene que tomar un branch, el hardware chequea si el branch está en el buffer; si es así, la próxima instrucción se levanta del buffer.
- Branch Prediction: existen varias técnicas para predecir si se va a tomar cierto branch. Algunas de las más comunes son:
  - Predict Never Taken
  - Predict Always Taken
  - Predict by Opcode
  - Taken/not Taken switch
  - Branch History Table

Los tres primeros enfoques son estáticos: no dependen de la ejecución del programa al momento de ejecutar la instrucción condicional.

En Predict Never Taken, el procesador asume por default que el salto nunca se toma, es decir que continúa levantando las instrucciones siguientes a las del salto. Funciona bien cuando el salto es “hacia adelante”.

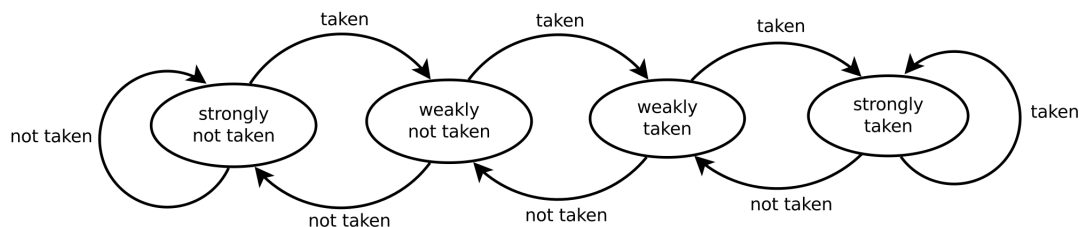
En Predict Always Taken, el procesador asume que el salto se toma siempre, es decir levanta las instrucciones a partir de la dirección target.

En Predict by Opcode, el procesador asume que el branch va a ser tomado para ciertos opcodes de branch y no para otros.

En los enfoque dinámicos, la idea es aumentar la cantidad de aciertos guardando información sobre las instrucciones de branch condicional, por ejemplo con 1 o más bits. Estos bits son referidos como taken/not taken switches que hacen que el procesador tome una decisión particular la próxima vez que encuentre la instrucción.

Con 1 solo bit, todo lo que se puede registrar es si la última ejecución de la instrucción resultó en un branch o no. Una limitación que tiene es que si un salto siempre resulta taken y falla una vez, produce dos predicciones fallidas seguidas, ya que el bit se invierte.

Para superar esta limitación, se utilizan 2 bits, generando un proceso de decisión que se puede representar como una máquina de estados finitos con 4 estados



En la práctica este tipo de branch prediction se implementa en la etapa de instruction fetch del pipeline, como un pequeño cache de direcciones. Otra forma de implementarlo es agregar un par de bits a cada bloque de líneas en el cache de instrucciones, que se emplean únicamente si ese bloque tiene instrucciones de salto condicional.

## 7.2. Hazards

En pipelining hay situaciones donde la siguiente instrucción no se puede ejecutar en el siguiente ciclo. Estas situaciones se llaman hazards, y existen 3 tipos:

- **Hazard estructural:** ocurre cuando el hardware no soporta la combinación de instrucciones que queremos ejecutar en un mismo ciclo de reloj. Ej: accesos concurrentes a memoria.
- **Hazard de datos:** ocurre cuando se para el pipeline debido a que una etapa debe esperar a que otra termine. Por ejemplo, si se hace una suma que se guarda en un registro, y luego se utiliza ese registro para hacer una resta. Para mitigar este problema se utiliza una técnica llamada forwarding, que consiste en extraer el resultado directamente de la salida de la unidad de ejecución y enviarlo a la entrada de la etapa que lo requiere.
- **Hazard de control:** ocurre cuando se tiene que tomar una decisión basada en el resultado de una instrucción mientras otras se están ejecutando.

## 8. Procesadores Superescalares

Un procesador superescalar es un procesador que utiliza múltiples pipelines independientes para ejecutar las instrucciones. Estos procesadores explotan lo que se llama paralelismo a nivel de instrucción, que se refiere al grado en que las instrucciones de un programa se pueden ejecutar en paralelo.

Otro enfoque para mejorar la performance es el **superpipelining**, en donde las etapas del pipeline se dividen en etapas más pequeñas, y al hacer cada etapa menos trabajo, se puede aumentar la velocidad del clock.

Estos procesadores tienen desventajas similares a las que surgen con pipelining:

- Data Hazards: se da en situaciones como la siguiente:

*add r1, r2*

*add r3, r1*

La segunda instrucción se puede levantar y decodificar pero no se puede ejecutar hasta que termine la primera instrucción. Para mitigar esto se utilizan técnicas como register renaming y out of order execution.

- Structural Hazards: se da cuando el procesador no tiene suficientes recursos para ejecutar varias instrucciones a la vez. Por ejemplo:

*add r1, r2*

*add r3, r4*

El procesador debe tener la capacidad de realizar 2 writes a la vez.

- Control Hazards: se dan cuando el procesador llega a un branch, y tiene que decidir qué instrucción levantar. Se resuelve con branch prediction.

### 8.1. Out of Order Execution

Con OoOE, el procesador ejecuta las instrucciones según la disponibilidad de los datos de entrada, y no en el orden original del programa. En este paradigma, el procesamiento de las instrucciones se divide en las siguientes etapas:

- Fetch de instrucciones.
- Envío de instrucciones a un buffer de instrucciones (también llamado *reservation station*).
- Las instrucciones esperan en el buffer hasta que sus operandos estén disponibles.
- Las instrucciones se despachan a la unidad funcional adecuada y se ejecutan.
- Los resultados se encolan.
- Solo después de que todas las instrucciones previas escriben sus resultados en el file register, este resultado es también escrito en el file register. Esta etapa se llama retire.

Para implementar un procesador con OoOE, necesitamos dividir la etapa de decodificación de instrucciones en dos sub etapas:

- La etapa de envío, que trabaja en orden, decodificando las instrucciones y enviándolas a la unidad de ejecución. Si encuentra un obstáculo estructural se detiene hasta que se resuelva el obstáculo.
- La etapa de lectura de operandos inicia la operación fuera de orden, ya que es capaz de saltar en el buffer aquellas instrucciones que tienen bloqueo de datos. Envía a ejecutar aquellas cuyos operandos pueden ser accedidos, y permanece revisando a que se resuelvan los obstáculos de datos para enviar esas instrucciones a ejecutar.

### 8.2. Scoreboarding

Es el método más sencillo para implementar OoOE evitando los riesgos asociados (Write After Read Y Write After Write). Las instrucciones pasan por las siguientes etapas:

- Unidad de envío: si hay una unidad funcional libre que pueda ejecutar la instrucción que se termina de decodificar, y no hay ninguna otra instrucción activa que necesite el mismo operando destino, entonces envía la instrucción a la unidad de ejecución. Luego actualiza la estructura de datos interna. Si aparece algún bloqueo estructural o detecta un riesgo WAW, entonces la unidad de envío se bloquea.



- Unidad de lectura de operando: el scoreboard monitorea para cada instrucción la disponibilidad de cada operando. Cuando una instrucción tiene todos sus operandos libres, el scoreboard envía una señal a la unidad funcional que tiene la instrucción, y comienza su ejecución. De este modo se asegura que no existan riesgos WAR. Esta unidad, junto con la de envío, reemplazan a la unidad de decodificación de un pipeline clásico.
- Unidad de ejecución: completa la ejecución de la instrucción y avisa al scoreboarding que la instrucción se ha completado. Equivale a la unidad de ejecución de un pipeline clásico.
- Unidad de escritura de resultado: una vez que la unidad funcional envió la señal al scoreboard, este se asegura de que no exista un riesgo WAR antes de escribir el resultado en el operando destino; caso contrario bloquea a la unidad de escritura.

Algunas limitaciones del scoreboarding es que aparecen nuevos obstáculos estructurales debido a que la cantidad de buses es limitada.

### 8.3. Tomasulo

Es otro algoritmo para implementar OoOE. Utiliza archivos de registros internos agrupados en uno o más bloques llamados *Reservation Stations*. Estos se encargan de buscar los operandos ni bien estén disponibles almacenándolos en estos registros internos. A medida que se emiten instrucciones, por cada operando pendiente se renombra el registro que lo contiene a un registro de la reservation station, es decir, las RS son un mecanismo para implementar register renaming. Una vez disponible el operando, la RS se encarga de su búsqueda y aplicación en cuanto registro destino lo necesite. De este modo, no hay posibilidad de que una instrucción cuya ejecución se adelanta respecto de otra previa en el programa, pueda modificar o utilizar un registro de la instrucción previa y que ésta luego use una copia incorrecta del mismo.

Para implementar el modelo de Tomasulo se requiere de dos bloques en el pipeline que trabajen de la siguiente manera:

- Etapas de envío: se encarga de obtener instrucciones desde una cola de prebúsqueda, tratando de ubicarlas en una RS vacía. Si no hay RS disponible, la instrucción se bloquea. Si los operandos no están disponibles en los registros, se rastrea cuales son las unidades de ejecución que los producirán y actualiza sus estructuras internas con esta información. Esto evita riesgos de WAR y WAW, ya que los registros se renombran.
- Etapas de ejecución: en caso de que los operandos no estén disponibles, queda a la espera de su generación, momento en el cual los copia a los registros internos de la RS que contiene a la instrucción.  
En general el orden en que se ejecutan es arbitrario, ya que los registros están renombrados. Esto es cierto salvo para las instrucciones de carga y almacenamiento en memoria.
- Etapas de escritura de resultado: en caso de que los operandos no estén disponibles, queda a la espera de su generación, momento en el cual los copia a los registros internos de la RS que contiene la instrucción.

## 9. Bibliografía

- Computer Organization and Architecture, Stallings (8th Edition): Capítulos 4, 17 y 18.
- Inside the Machine, Stokes: Capítulos 3 y 4.
- Clases teóricas de Furfaro.